Raisonnement à contraintes pour le test de bytecode Java

Florence Charreteur, Arnaud Gotlieb

Raisonnement à contraintes pour le test de bytecode Java

Florence Charreteur * **Arnaud Gotlieb** ‡

*Université de Rennes 1/IRISA, ‡INRIA/IRISA
Campus de Beaulieu, 35042 Rennes Cedex, France
{fcharret,gotlieb}@irisa.fr

Résumé

Le test logiciel permet d'augmenter la confiance que l'on porte à un programme ou un système. Dans ce contexte, il s'agit d'exécuter le programme avec un certain nombre d'entrées dans le but de couvrir des objectifs de test, comme celui qui consiste à atteindre toutes les instructions du programme au moins une fois durant la phase de test. Idéalement, on souhaite générer ces entrées de manière automatique mais ce problème est indécidable dans le cas général. Dans notre travail, nous avons développé une méthode (incomplète) pour ce problème qui s'appuie sur une vision relationnelle du programme. Cet article présente des opérateurs à contraintes servant à modéliser les instructions du bytecode Java sous la forme d'une relation entre deux états de la mémoire : l'état de la mémoire avant l'instruction et l'état de la mémoire après l'instruction. Les opérateurs expriment des liens entre des états du tas, celui-ci est vu comme une fonction définie sur une partie des entiers naturels. Les algorithmes de filtrage de ces opérateurs permettent des déductions fortes que nous illustrons sur un exemple complexe. Ce modèle est, à notre connaissance, le premier modèle à contraintes proposé pour tester le bytecode Java.

1 Introduction

Dans le contexte du test logiciel, il est utile de pouvoir trouver automatiquement des valeurs pour les paramètres d'une méthode permettant de satisfaire un objectif tel qu'atteindre une instruction particulière du programme ou encore obtenir certaines valeurs en sortie de méthode [2, 7]. L'usage de la programmation par contraintes fut introduite il y a quinze ans, dans le cadre du test par mutation [8]. Depuis, le test à base de contraintes s'est développé, et plusieurs générateurs automatiques de cas de test implémentent cette approche [2, 13, 12]. Le principe est d'extraire du code source un système de contraintes, et d'exploiter la propagation de contraintes et le backtracking pour trouver des données de test qui couvrent un élément sélectionné (chemin, instruction, branche) du code. Si le système de contraintes est montré insatisfiable, alors l'élément sélectionné ne peut pas être exécuté. Nous nous intéressons à la modélisation par contraintes de programmes en bytecode Java et non pas en code source Java. En effet, si un programme est réutilisé, ou réalisé par un sous-traitant, son code source n'est pas nécessairement disponible. Dans le cadre des systèmes critiques, il est souhaitable d'avoir exécuté toutes les instructions du programme au moins une fois, y compris pour le code sous-traité ou réutilisé. De plus, pour tester un programme qui fait appel à des librairies dont le code source n'est pas disponible, il est profitable de pouvoir également modéliser le code de ces librairies.

Dans le modèle proposé, une instruction bytecode est exprimée sous la forme d'une relation qui lie deux états de la mémoire. Un état de la mémoire est défini par la quantité de mémoire allouée et par la valeur contenue dans chacune des zones de mémoire allouées. Le tas est la zone de la mémoire destinée à stocker les objets alloués dynamiquement. Des opérateurs portant sur des fonctions ont été conçus pour exprimer les relations entre deux états du tas. Ces opérateurs permettent l'ajout d'un élément dans le tas, l'accès à une valeur stockée dans le tas et la mise à jour d'une valeur du tas. Les fonctions peuvent être modélisées par un ensemble de couples *(antécédent,image)*, les opérateurs portant sur les fonctions sont en cela proches des opérateurs ensemblistes ($\in$, $\subseteq$, $\cup$, etc.). Ces ensembles de couples sont cependant particuliers, car ce ne sont pas les couples qui doivent être distincts entre eux mais seulement les antécédents. De plus,

la cardinalité des ensembles est une variable. Les solveurs ensemblistes existants ne traitent pas, à notre connaissance, de tels ensembles. C'est pourquoi, nous nous sommes attachés à proposer un modèle relationnel du bytecode sous forme d'opérateurs à contraintes spécifiques. La définition de ce modèle constitue la principale contribution de ce papier.

Cet article est structuré de la manière suivante : la section 2 donne les notations, le type de programmes traités, et décrit les étapes de l'exécution d'un programme en bytecode Java ; la section 3 illustre comment un programme en bytecode Java peut être modélisé par des contraintes ; la section 4 expose le problème de la détermination de la forme de la mémoire ; la section 5 définit le domaine des variables sous contraintes représentant des fonctions, elle donne aussi les algorithmes de filtrage des trois opérateurs proposés ; la section 6 présente les travaux connexes.

2 Contexte

2.1 Notations

Dans cet article, nous utilisons des notations sur les séquences et sur les fonctions. Une séquence vide est notée ϵ, l'ajout d'un élément v en tête d'une séquence s est noté $v.s$. L'accès à l'élément d'index n d'une séquence est notée $s[n]$. Les index de séquence commencent à 0. Une fonction h est décrite en extension par l'ensemble des couples $(x, h(x))$. La mise à jour de l'image associée à l'antécédent x est notée $h[x \mapsto v]$ où v est la nouvelle valeur associée. Les noms des variables de notre modèle débutent par une majuscule. Les variables désignant des entiers et des références sont indicées respectivement par i et r. Les valeurs inconnues sont notées $\_$. $dom(V)$ désigne le domaine de la variable V.

2.2 Cadre de l'article

Nous nous concentrons dans cet article sur les instructions du bytecode Java d'accès et de mise à jour de la mémoire, en particulier pour la gestion dynamique de la mémoire. Les instructions avec transfert de contrôle sont pour l'instant laissées de côté : bien que les constructions conditionnelles et les boucles soient déjà traitées dans le contexte du langage C [2], ce traitement devra être adapté au bytecode qui est destructuré. Le mécanisme d'appel de méthode n'est pas décrit : seules les zones de la mémoire utilisées pour l'exécution de la méthode à laquelle appartient le bloc d'instructions considéré sont prises en compte. Le multithreading n'est pas supporté par notre modèle car nous faisons l'hypothèse que les données manipulées par le programme ne sont pas modifiées par d'autres threads. Les variables flottantes sont laissées de côté pour l'instant, leur cas pourrait être traité en utilisant la méthode décrite dans [5]. Le ramasse miettes désalloue les zones de la mémoire auxquelles le programme n'accède plus. Il n'est pas modélisé car son activation n'affecte pas le résultat de l'analyse sauf en présence de méthodes `finalize`. Le cas où des erreurs ou des exceptions seraient levées n'est pas étudié : nous modélisons une sémantique du bytecode sans erreurs et sans exceptions.

2.3 Description du fonctionnement de la JVM

Le bytecode Java est la représentation compilée des programmes Java. La machine virtuelle Java (JVM) interprète le bytecode. La description et la modélisation des instructions se basent sur la spécification des instructions donnée par Sun [11]. Le lecteur familier du bytecode Java peut sans problème passer à la section 3.

2.3.1 Zones de stockage des données dans l'environnement de la JVM.

L'exécution d'un programme en bytecode par la JVM se fait dans un environnement. Cet environnement comprend notamment des zones de la mémoire pour stocker les données lors de l'exécution du programme. Les types de données manipulées par la JVM sont les **entiers**, les **flottants**, et les **références**. Dans cet article, les références sont des adresses de la mémoire où sont stockés les objets. L'environnement comprend pour chaque méthode des **registres**[1] et une **pile d'opérandes**. Le nombre de registres est fixe et connu, de même que la taille maximale de la pile d'opérandes. Les registres stockent la référence de l'objet sur laquelle s'applique la méthode (this) si la méthode n'est pas statique, les paramètres de la méthode et les variables locales à la méthode. Les objets, alloués dynamiquement (instruction Java `new`), sont stockés dans le **tas**. Le tas est une zone de la mémoire partagée entre toutes les méthodes. Les accès à ces objets du tas se font via des références. Dans cet article, on appelle état de la mémoire l'état des registres (les valeurs stockées), de la pile (la taille de la pile et les valeurs empilées) et du tas (la forme du tas et les valeurs stockées).

2.3.2 Effet de l'exécution des instructions sur la mémoire.

Quand les instructions bytecode sont exécutées par la JVM, l'état de la mémoire se modifie. Ce changement d'état de la mémoire au cours d'une exécution

[1] variables locales dans la spécification Sun

est illustré sur un exemple. Le code du listing 1 est celui d'une classe `Coord` représentant les coordonnées cartésiennes d'un point : `x` et `y`. La méthode `deplaceY` permet de créer de nouvelles coordonnées pour exprimer un déplacement caractérisé par un temps écoulé, donné par un objet de type `Chrono`, et une vitesse entière. Le résultat de la compilation de cette méthode en bytecode Java est donné sous une forme lisible (après désassemblage par la commande javap du JDK) sur le listing 2.

```
public class Coord {
 int x,y;

 public Coord(int cx, int  cy){x=cx; y=cy;}

 public Coord deplaceY(Chrono chrono,int vitesse){
    int ytemp=y+chrono.temps*vitesse;
    chrono.temps=0;
    return new Coord(x,ytemp);}
 }

 public class Chrono {
    public int temps;
  ...
 }
```

Listing 1 – code Java de la classe Coord qui modélise des coordonnées

```
public Coord deplaceY(Chrono, int) ;
Code :
 Stack=4, Locals=4, Args_size=3
 0 :  aload_0      //empile le contenu du registre 0
 1 :  getfield #3//remplace le sommet de pile
                   par la valeur de l'attribut y
 4 :  aload_1      //empile le contenu du registre 1
 5 :  getfield #4//remplace le sommet de pile par
                   la valeur de l'attribut temps
 8 :  iload_2      //empile le contenu du registre 2
 9 :  imul         //multiplie les deux éléments de sommet
                   de pile et les remplace par le résultat
 10 : iadd         //additionne les deux éléments de sommet
                   de pile et les remplace par le résultat
 11 : istore_3     //stocke le sommet de pile dans
                   le registre 3
 12 : aload_1      //empile le contenu du registre 1
 13 : iconst_0     //empile la constante 0
 14 : putfield #4//modifie la valeur de l'attribut temps
 17 : new #5 ;     //réserve de la place en mémoire
                   pour un objet de type Coord
 20 : dup          //duplique le sommet de pile
 21 : aload_0      //empile le contenu du registre 0
 22 : getfield #2//remplace le sommet de pile par
                   la valeur de l'attribut x
 25 : iload_3      //empile le contenu du registre 3
 26 : invokespecial #6 ;//invoque le constructeur
 29 : areturn      //retourne le sommet de pile
```

Listing 2 – bytecode Java de la méthode deplaceY

L'état de la mémoire en entrée d'une méthode donne la valeur des paramètres lors de l'appel de cette méthode. Les états de la mémoire en deux points de programme consécutifs sont liés par l'instruction séparant ces points de programme. Nous allons décrire l'état de la mémoire en chaque point de programme au cours d'une exécution. Dans cette description, les registres f sont vus comme une séquence. La pile d'opérandes s est également vue comme une séquence, dont le premier élément donne le sommet de pile. Le tas est vu comme une fonction h qui associe aux adresses mémoires des zones allouées la valeur qui y est stockée. Dans notre description un objet est stocké sur autant de zones mémoire consécutives qu'il possède d'attributs. Si un objet est stocké à l'adresse a, son nième attribut est à l'adresse a+n-1. Pour une classe, notre représentation associe à chaque attribut un numéro unique qui caractérise sa position relative dans la mémoire. Pour la classe `Coord`, on associe l'index 0 à l'attribut `x` et l'index 1 à l'attribut `y`. Pour la classe `Chrono`, on associe l'index 0 à l'attribut `temps`. Dans cette description, une valeur est stockée sur une seule zone de la mémoire quelle que soit sa taille. L'état de la mémoire avant une instruction numérotée j de la méthode `deplaceY` sera notée $m_j = (f, s, h)$.
Au début de l'exécution de la méthode `deplaceY`, les registres contiennent quatre valeurs : $f[0]$ donne l'adresse du tas où est stocké `this`, $f[1]$ donne l'adresse du tas où est stocké l'objet `chrono`, $f[2]$ donne la valeur du paramètre `vitesse` et $f[3]$ donne la valeur de la variable `ytemp` locale à la fonction. La pile est vide. Le tas comprend deux objets : l'objet référencé par `this` dont les valeurs des attributs `x` et `y` sont stockés respectivement aux adresses données par $f[0]$ et $f[0]+1$, et l'objet référencé par `chrono` dont la valeur de l'attribut `temps` est stockée à l'adresse donnée par $f[1]$.Voici l'évolution de l'état de la mémoire au cours de l'exécution de cette méthode :

Etape1
$m_0 = (f[0].f[1].f[2].f[3], \epsilon, h)$
$m_1 = (f[0].f[1].f[2].f[3], f[0], h)$
$m_4 = (f[0].f[1].f[2].f[3], h(f[0]+1), h)$
$m_5 = (f[0].f[1].f[2].f[3], f[1].h(f[0]+1), h)$
$m_8 = (f[0].f[1].f[2].f[3], h(f[1]+0).h(f[0]+1), h)$
$m_9 = (f[0].f[1].f[2].f[3], f[2].h(f[1]+0).h(f[0]+1), h)$
$m_{10} = (f[0].f[1].f[2].f[3], f[2] * h(f[1]+0).h(f[0]+1), h)$
$m_{11} = (f[0].f[1].f[2].f[3], exp, h)$
avec $exp = f[2] * h(f[1]+0) + h(f[0]+1)$
$m_{12} = (f[0].f[1].f[2].exp, \epsilon, h)$
Etape 2
$m_{13} = (f[0].f[1].f[2].exp, f[1], h)$
$m_{14} = (f[0].f[1].f[2].exp, 0.f[1], h)$
$m_{17} = (f[0].f[1].f[2].exp, \epsilon, h[f[1]+0 \mapsto 0])$
Etape 3
$m_{20} = (f[0].f[1].f[2].exp, a,$
$h[f[1]+0 \mapsto 0, a \mapsto \_, a+1 \mapsto \_])$
$m_{21} = (f[0].f[1].f[2].exp, a.a,$
$h[f[1]+0 \mapsto 0, a \mapsto \_, a+1 \mapsto \_])$
$m_{22} = (f[0].f[1].f[2].exp, f[0].a.a,$
$h[f[1]+0 \mapsto 0, a \mapsto \_, a+1 \mapsto \_])$
$m_{25} = (f[0].f[1].f[2].exp, h(f[0]+0).a.a,$
$h[f[1]+0 \mapsto 0, a \mapsto \_, a+1 \mapsto \_])$
$m_{26} = (f[0].f[1].f[2].exp, exp.h(f[0]+0).a.a,$
$h[f[1]+0 \mapsto 0, a \mapsto \_, a+1 \mapsto \_])$

$m_{29} = (f[0].f[1].f[2].exp, a,$
$h[f[1] + 0 \mapsto 0, a \mapsto exp, a + 1 \mapsto h(f[0] + 0)])$

On distingue 3 étapes dans l'exécution : le calcul de la valeur affectée à `ytemp`, la mise à 0 du champ `temps` de l'objet référencé par `chrono` et la création du nouvel objet de type `Coord`.
Etape 1. L'instruction 0 charge sur la pile l'adresse `this`, stockée dans le premier registre *f[0]*. L'instruction 1 remplace le sommet de pile par la valeur du champ `y` de l'objet référencé par `this`. L'instruction 4 charge l'adresse de `chrono` sur la pile. L'instruction 5 remplace le sommet de pile par la valeur du champ `temps` de l'objet référencé par `chrono`. L'instruction 8 charge depuis le registre 2 le paramètre `vitesse` sur la pile. L'instruction 9 remplace les deux éléments en sommet de pile par le résultat de leur multiplication. L'instruction 10 procède de manière similaire avec l'addition. A cette étape, la valeur en sommet de pile vaut `y`+`chrono.temps`*`vitesse`, valeur notée *exp*. L'instruction 11 stocke cette valeur dans `ytemp`.
Etape 2. L'instruction 12 charge sur la pile l'adresse de `chrono`, et l'instruction 13 empile la constante 0. L'instruction 14 stocke la valeur du sommet de pile dans l'attribut `temps` de l'objet référencé par `chrono`.
Etape 3. Par l'instruction 17, une zone est réservée dans le tas pour stocker un nouvel objet de type `Coord`, et l'adresse à laquelle sera stocké l'objet est empilée. Ici, l'adresse vaut a et deux zones sont réservées dans le tas pour stocker l'objet : l'une à l'adresse a pour stocker le premier attribut (`x`) et l'autre à l'adresse $a + 1$ pour stocker le deuxième attribut (`y`). Aucune valeur n'est donnée à ces nouveaux attributs. L'instruction 20 duplique le sommet de pile. L'instruction 21 charge l'adresse de `this` et l'instruction 22 remplace cette adresse par la valeur de l'attribut `x` de l'objet référencé par `this`. L'instruction 25 charge la valeur de la variable locale 3 (`ytemp`). L'instruction 26 appelle le constructeur qui initialise les valeurs des attributs de l'objet `Coord` nouvellement créé. Ce mécanisme d'appel de méthode n'est pas détaillé, seul son effet est montré : le sommet de pile (`ytemp`) est associé à l'adresse du deuxième attribut, et `this.x`, lui aussi dans la pile, est associé à l'adresse du premier attribut. En sortie de méthode, l'instruction 29 enlève l'adresse a du sommet de pile, et la retourne comme résultat de la méthode. Cette adresse est celle de l'objet nouvellement créé.

2.3.3 Instructions bytecode traitées

Les instructions bytecode traitées dans le cadre de cet article sont listées ci-dessous. La première lettre de certaines des instructions indique le type de valeurs manipulées par l'instruction : `i` pour `int`, `l` pour `long`, `a` pour référence.

- `iconst_`$\langle c \rangle$, `lconst_`$\langle c \rangle$, `aconst_null` pour charger la constante c ou `null` sur la pile.
- `iload_`$\langle n \rangle$, `lload_`$\langle n \rangle$, `aload_`$\langle n \rangle$ pour charger la valeur stockée dans le nième registre sur la pile.
- `istore_`$\langle n \rangle$, `lstore_`$\langle n \rangle$, `astore_`$\langle n \rangle$ pour stocker la valeur de sommet de pile dans le nième registre.
- `dup` pour dupliquer le sommet de pile.
- T`add`, T`sub`, T`mul`, T`div`, T`rem` où T vaut `i` ou `l`, qui appliquent une opération binaire sur les deux éléments en sommet de pile.
- `new #c` qui crée un nouvel objet. `#c` donne la classe de l'objet.
- `getfield #v` qui accède à un attribut de l'objet référencé par la valeur en sommet de pile. `#v` indique à quel attribut on accède. La référence en sommet de pile est dépilée, l'attribut est empilé.
- `putfield #v` qui stocke la valeur de sommet de pile dans un attribut de l'objet référencé par la deuxième valeur dans la pile. `#v` indique quel attribut est mis à jour.
- `invokespecial #m` pour appeler une méthode particulière (dans notre cadre un constructeur) avec pour paramètres les valeurs en sommet de pile. `#m` donne la méthode à invoquer.

3 Modèle à contraintes de la JVM

3.1 Intérêt du modèle

Le but de notre modélisation pour le bytecode Java est de pouvoir générer des tests face à des objectifs de test. Un problème qui se pose en test est le suivant : si l'on veut satisfaire des hypothèses portant sur la valeur de retour et sur la valeur des paramètres à la fin de l'exécution de la méthode, avec quels paramètres appeler la méthode pour satisfaire ces hypothèses ? Autrement dit, quel est le lien qui existe entre l'état de la mémoire en sortie de méthode et l'état de la mémoire en entrée de méthode (l'état de la mémoire en entrée donnant la valeur des paramètres). Ainsi, sur le programme Java du listing 1, on peut se demander quelle valeur donner aux paramètres `chrono` et `vitesse` pour obtenir une coordonnée `y` qui dépasse une ordonnée fixée. On pourrait se poser la même question sous l'hypothèse que la vitesse est inférieure à une valeur fixée. Un autre objectif de test est celui de couvrir toutes les branches d'une méthode. L'un des problèmes rencontrés est la détermination de valeurs pour les paramètres en entrée de méthode tels qu'une instruction conditionnelle soit vraie. C'est à dire qu'il faut déterminer un état de la mémoire en entrée de méthode tel que l'état de la mémoire avant l'instruction conditionnelle rende cette instruction conditionnelle vraie. Ce cas n'est pas traité dans l'article, car, comme précisé

dans le cadre du papier, on ne prend pas pour l'instant en compte les instructions conditionnelles. Mais l'objectif de notre modélisation est aussi de pouvoir répondre à terme à ce genre d'objectif.
Pour répondre à ces objectifs, l'effet de l'exécution de chaque instruction bytecode est exprimé sous la forme d'une relation entre deux états de la mémoire : l'état de la mémoire avant l'exécution de l'instruction et l'état de la mémoire après l'exécution de l'instruction. Ces relations modélisant les instructions, ainsi que des contraintes représentant l'objectif de test à atteindre, permettent d'obtenir un système à contraintes. Une solution de ce système à contraintes est un état mémoire en entrée satisfaisant l'objectif de test.

3.2 Modèle proposé

Pour modéliser les instructions, il faut d'abord modéliser les données manipulées par la JVM et les zones de stockage de ces données. Les variables entières manipulées par le programme sont des variables entières à domaine fini dans notre modèle. Leur domaine initial dépend du type de l'entier (`int`, `long`,...) dans le programme. Les références manipulées par le programme sont représentées dans notre modèle par des variables dont le domaine peut être considéré comme un ensemble d'adresses qui désignent des zones mémoires du tas vers lesquelles la référence peut pointer. Les zones de stockage des données sont les registres, la pile d'opérandes et le tas. Les registres et la pile sont représentés par des séquences de variables. Le tas est représenté par une variable qui modélise une fonction dont le domaine de définition est une partie de $\mathbb{N}$. Cette fonction associe à chaque adresse (représentée par un entier) la valeur qui lui est associée dans le tas. Le domaine de cette variable sera détaillé dans la suite ainsi que des opérateurs portant sur ce type de variables.
Soit $M_j = (F_j, S_j, H_j)$ l'état de la mémoire avant l'instruction j, avec F_j les registres, S_j la pile d'opérandes et H_j le tas. En modélisant les instructions de la méthode `deplaceY` du listing 2, on obtient le système à contraintes suivant (là encore, par souci de simplicité, l'appel au constructeur n'est pas détaillé, son effet est directement pris en considération) :

$\{M_0 = (F, \epsilon, H),\ F = This_r.Chrono_r.Vit_i.Ytemp_i$
$M_1 = (F, This_r, H),$
$M_4 = (F, Y_i, H),\ This_r \neq null,\ Thisy_r = This_r + 1,$
$\quad (Thisy_r, Y_i) \in H,$
$M_5 = (F, Chrono_r.Y_i, H),$
$M_8 = (F, Temps_i.Y_i, H),\ Chrono_r \neq null,$
$\quad (Chrono_r, Temps_i) \in H,$
$M_9 = (F, Vit_i.Temps_i.Y_i, H),$
$M_{10} = (F, Mul_i.Y_i, H),\ Mul_i = Vit_i * Temps_i,$
$M_{11} = (F, Add_i, H),\ Add_i = Mul_i + Y_i,$
$M_{12} = (F2, \epsilon, H),\ F2 = This_r.Chrono_r.Vit_i.Add_i,$
$M_{13} = (F2, Chrono_r, H),$
$M_{14} = (F2, 0.Chrono_r, H),$
$M_{17} = (F2, \epsilon, H1),\ Chrono_r \neq null,\ H1 = H\,[Chrono_r \mapsto 0],$
$M_{20} = (F2, A_r, H3),\ A_r \neq null,\ H2 = H1 \sqcup \{(A_r, \_)\},$
$\quad A1_r = A_r + 1,\ H3 = H2 \sqcup \{(A1_r, \_)\},$
$M_{21} = (F2, A_r.A_r, H3),$
$M_{22} = (F2, This_r.A_r.A_r, H3),$
$M_{25} = (F2, X_i.A_r.A_r, H3),\ This_r \neq null,\ (This_r, X_i) \in H3,$
$M_{26} = (F2, Add_i.X_i.A_r.A_r, H3),$
$M_{29} = (F2, A_r, H5),\ H4 = H3\,[A_r \mapsto X_i],\ A2_r = A_r + 1,$
$\quad H5 = H4\,[A2_r \mapsto Add_i]\}$

Les trois principales instructions de manipulation des objets dans le tas sont `getfield`, `putfield` et `new`, et se traduisent par des opérateurs d'accès, de mise à jour et d'ajout.
`getfield`. Si l'instruction j est `getfield #v,` où `#v` désigne un champ d'index n, on pose les contraintes suivantes qui lient M_j et M_{j+1} :
$\{M_j = (F, Ref_r.S, H),\ M_{j+1} = (F, Val.S, H),\ Ref_r \neq null,$
$RefField_r = Ref_r + n,\ (RefField_r, Val) \in H\}$
La contrainte $(RefField_r, Val) \in H$, avec $\in$ l'opérateur d'accès, indique qu'à l'adresse référencée par $RefField_r$ du tas modélisé par H (H représente une fonction) est associée la valeur Val.
`putfield`. Si l'instruction j est `putfield #v`, où `#v` désigne un champ d'index n, on pose les contraintes suivantes qui lient M_j et M_{j+1} :
$\{M_j = (F, Val.Ref_r.S, H),\ M_{j+1} = (F, S, H1),\ Ref_r \neq null,$
$RefField_r = Ref_r + n,\ H1 = H\,[RefField_r \mapsto Val]\}$
La contrainte $H1 = H\,[RefField_r \mapsto Val]$, avec $H[A \mapsto B]$ l'opérateur de mise à jour, indique que le tas modélisé par $H1$ est le même que le tas modélisé par H hormis que la valeur stockée à l'adresse référencée par $RefField_r$ a été mise à jour par Val.
`new`. Si l'instruction j est `new #c`, instruction qui réserve de la place pour stocker un objet d'une classe donnée par `#c`, on pose plusieurs contraintes. Chacune d'entre elles utilise l'opérateur d'ajout $\sqcup$ pour réserver de la place pour un attribut. Les adresses des zones réservées en mémoire pour stocker les attributs sont contiguës. La contrainte $H2 = H1 \sqcup \{(A_r, \_)\}$ indique que le tas modélisé par $H2$ est le même que le tas modélisé par $H1$ avec une zone de mémoire en plus à l'adresse référencée par A_r. Ici, la valeur stockée est inconnue.

Ces trois opérateurs à contraintes pour manipuler le tas prennent donc en paramètres notamment des variables modélisant des fonctions et des variables modélisant des références.

4 Problème de la détermination de la forme de la mémoire

La forme de la mémoire en entrée d'une méthode n'est pas toujours connue : la quantité de mémoire allouée pour stocker les paramètres peut ne pas être déterminée. Prenons le cas d'une classe *Maillon* per-

mettant de décrire une liste chaînée grâce à deux attributs : un attribut *valeur* de type entier et un attribut *suivant* de type *Maillon*. Si l'on veut modéliser par contraintes le comportement d'une méthode prenant en paramètre un *Maillon* m, on ne veut pas faire d'hypothèses sur la valeur de m. Ainsi, on ne sait pas si le champ *suivant* de m vaut `null`, ou s'il référence un autre maillon dont la valeur est elle aussi indéterminée. Dans notre modélisation, la valeur du *Maillon* m passé en paramètre est stockée dans le tas, et l'attribut *suivant* de m est représenté par une variable référence N_r. Comme aucune information sur sa valeur n'est disponible, on notera $dom(N_r) = all$ ce qui signifie que N_r peut référencer n'importe quelle zone de la mémoire ou valoir `null`. Comme le nombre d'éléments en mémoire est inconnu, la mémoire en entrée de méthode est dite non close. Des éléments sont en effet susceptibles d'être ajoutés en mémoire selon l'hypothèse qui sera faite sur la valeur de *m.suivant* lors de la phase d'énumération. Un autre cas où la mémoire peut être non close est celui d'une boucle contenant une allocation dynamique. Si aucune déduction n'est possible sur le nombre de fois que la boucle est dépliée, au point de programme suivant la boucle le nombre d'objets alloués en mémoire est inconnu : la mémoire n'est pas close. Le domaine d'une variable référence vaut donc soit *all*, soit un ensemble énuméré comportant la valeur *null* et/ou des entiers (l'ensemble des adresses pouvant être référencées).
De plus, des contraintes du type $N_r \neq a$ sont ajoutées au système, indiquant que la variable référence N_r ne peut pas référencer l'adresse a. Deux cas sont à distinguer. Soit le domaine de N_r vaut *all*, en ce cas N_r peut valoir n'importe quelle adresse mais pas les adresses a figurant dans de telles contraintes. Soit le domaine de N_r est énuméré et les adresses a figurant dans ce type de contraintes sont enlevées du domaine. De même, on peut exprimer qu'une référence doit être non nulle par la contrainte $N_r \neq null$. Le domaine d'une référence ne peut que rétrécir au cours de la propagation : le domaine *all* est le domaine le plus grand possible, les domaines énumérés sont d'autant plus grand que leur cardinal est grand.

5 Modélisation du tas : définition de relations entre des fonctions

5.1 Opérateurs nécessaires

Les états du tas sont représentés comme des fonctions ayant pour domaine une partie de $\mathbb{N}$ et pour codomaine une union d'ensembles (entiers, réels, symboles...). Une fonction f sera représentée en extension. L'objectif est de fournir des contraintes exprimant des relations entre deux fonctions. Les fonctions sont donc des variables du problème à résoudre. Trois opérateurs nous intéressent :

- l'ajout : $F1 = F2 \sqcup \{(Antecedent_r, Image)\}$,
- l'accès : $(Antecedent_r, Image) \in F$,
- la mise à jour : $F2 = F1\,[Antecedent_r \mapsto Image]$,

avec F, $F1$ et $F2$ des variables modélisant des fonctions, $Antecedent_r$ une variable modélisant une référence et $Image$ une variable typée. Ces opérateurs permettent de restreindre le domaine des états de la mémoire par propagation.

Illustration *Soit F1 une variable modélisant une fonction. La notation* $F1 = \{(1, A), (2, B), ...\}$ *indique que la fonction représentée par F1 est définie au moins sur* $\{1, 2\}$*, et potentiellement sur un domaine plus grand ce qui est signifié par les points de suspension. L'image de* 1 *par cette fonction prend sa valeur dans le domaine de la variable A, l'image de* 2 *dans le domaine de la variable B. Soit* $F1 = \{(1, A), (2, B), ...\}$, $F2 = \{(2, U), (3, 6)\}$, $dom(A) = 0..2$, $dom(B) = 1..4$, $dom(U) = 3..6$. *En posant la contrainte* $F1 = F2 \sqcup \{(1, Val)\}$, *avec* $dom(Val) = 0..1$, *on veut obtenir les déductions suivantes :* $F1 = \{(1, A), (2, B), (3, 6)\}$, $F2 = \{(2, U), (3, 6)\}$, $A = Val$, *d'où* $dom(A) = dom(Val) = 0..1$ *et* $B = U$ *d'où* $dom(B) = dom(U) = 3..4$. *Le domaine de* $F1$ *après déductions comporte 4 valeurs :* $\{(1, 0), (2, 3), (3, 6)\}$, $\{(1, 0), (2, 4), (3, 6)\}$, $\{(1, 1), (2, 3), (3, 6)\}$ *et* $\{(1, 1), (2, 4), (3, 6)\}$.

5.2 Des variables modélisant des fonctions

Domaine d'une variable modélisant une fonction. Le domaine d'une variable F modélisant une fonction est défini par deux éléments : un ensemble de couples et un statut.

1. L'ensemble de couples, noté E_F, contient les couples obligatoirement présents dans la fonction. Dans ces couples de la forme *(antécédent,image)*, les antécédents sont des constantes entières positives, les images sont des variables typées à domaine fini.
2. Le statut est un symbole qui vaut soit *non clos*, soit *clos*. Si le statut vaut *non clos*, aux couples de E_F sont susceptibles de s'ajouter d'autres couples. Si le statut vaut *clos*, aucun couple ne peut s'ajouter à ceux de E_F.

Les couples *(antécédent,image)* contenus dans E_F ont des antécédents tous distincts. Au cours du processus de résolution du système de contraintes, l'ensemble de couples E_F ne peut qu'être enrichi, réduisant ainsi le nombre de valeurs possibles pour la fonction.
Un statut *non clos* pour une variable représentant une fonction est à mettre en relation avec la présence de points de suspension dans les notations précédentes, et à la notion de mémoire non close décrite dans la section

4. Lorsque le statut associé à une variable représentant une fonction vaut *clos*, le domaine de définition de la fonction est intégralement connu et E_F ne peut plus être enrichi. Par contre il est possible que certains éléments du codomaine soient encore variables. Au cours du processus de résolution du système de contraintes, le statut ne peut que passer de *non clos* à *clos*.

Instanciation d'une variable modélisant une fonction. Une variable sous contrainte représentant une fonction est instanciée quand le statut associé à la variable vaut *clos*, et qu'à chaque élément du domaine de la fonction est associée par cette fonction une constante (le codomaine est intégralement connu).

5.3 Description des opérateurs

Pour chaque opérateur, les déductions opérées par le filtrage sont décrites sous forme de règles numérotées. Les algorithmes de filtrage sont fournis, les numéros figurant sur ces algorithmes font référence aux numéros de règles. Le filtrage opère en réduisant les domaines des variables sur lesquelles portent l'opérateur ou en ajoutant des contraintes dans le système de contraintes. Au cours de ce filtrage, si le domaine d'un entier ou d'une référence devient vide l'opérateur échoue. Il en va de même si l'ensemble de couples E_F d'une variable représentant une fonction F est enrichi alors que le statut de la fonction est clos.

5.3.1 Ajout

Soit la contrainte $F' = F \sqcup \{(Ref_r, Val)\}$ qui signifie que F' décrit la même fonction que F avec le couple (Ref_r, Val) en plus. Si cette contrainte figure dans le système de contraintes, il faut aussi que la variable référence Ref_r ne vaille pas *null*. L'algorithme de filtrage de l'opérateur d'ajout (algorithme 1) procède de la manière suivante.

1. Tous les éléments du domaine de définition de F doivent être dans le domaine de définition de F'.
2. Tous les éléments du domaine de définition de F' qui ne font pas partie du domaine de Ref_r doivent être dans domaine de définition de F.
3. Tout élément qui est à la fois dans le domaine de définition de F et dans celui de F' a la même image par F et F'.
4. Tout élément appartenant au domaine de définition de F doit être retiré du domaine de Ref_r
5. Pour tout élément a du domaine de Ref_r tel qu'il existe (a, I') dans F', si la contrainte $Val = I'$ n'a pas de solution a doit être retiré du domaine de Ref_r. En effet, si $Ref_r = a$, il faut que $Val = I'$. Donc $Ref_r = a$ est faux et a est retiré du domaine de Ref_r.
6. Si Ref_r est instancié et vaut a, a doit être ajouté au domaine de définition de F' s'il n'y est pas, et Val et l'image de a par F' sont égaux.
7. Si le statut de F' est clos, le domaine de Ref_r est inclus dans le domaine de définition de F'.
8. Si le statut de F' ou celui de F est clos, et que Ref_r est instancié, alors les statuts de F et F' sont tous les deux clos. En effet, si le statut de F ou de F' est clos, l'un des deux domaines de définition est connu. Si Ref_r est instancié, l'élément à ajouter au domaine de F pour obtenir le domaine de F' est également connu. Donc les deux domaines de définition sont connus.

La contrainte d'ajout est réveillée si l'ensemble de couples associé à F ou à F' est enrichi, si le statut de F ou de F' est modifié, si le domaine d'une variable du codomaine de F' bouge, ou encore si le domaine de Ref_r ou celui de Val bouge.

Algorithme 1 : Algorithme de filtrage de l'opérateur d'ajout

soient :
$F' = F \sqcup \{(Ref_r, Val)\}$ la contrainte à traiter
C le reste du système de contraintes
début
  pour chaque $(a, I') \in E_{F'}$ **faire**
    si $a \in dom(Ref_r)$ **alors**
      si $dom(I') \cap dom(Val) = \emptyset$ **alors**
        $C \longleftarrow C \cup (Ref_r \neq a)$ ***(5)***
    si $(a, I) \in E_F$ **alors**
      $C \longleftarrow C \cup (I = I')$ ***(3)***
    si $a \notin (dom(E_F) \cup dom(Ref_r))$ **alors**
      $E_F \longleftarrow E_F \cup (a, I')$ ***(2,3)***
  pour chaque $(a, I) \in E_F$ **faire**
    si $a \in dom(Ref_r)$ **alors**
      $C \longleftarrow C \cup (Ref_r \neq a)$ ***(4)***
    si $a \notin dom(E_{F'})$ **alors**
      $E_{F'} \longleftarrow E_{F'} \cup (a, I)$ ***(1,3)***
  si $statut_{F'} = clos$ **alors**
    $dom(Ref_r) \longleftarrow dom(Ref_r) \cap dom(E_{F'})$ ***(7)***
  si $ground(Ref_r)$ **alors**
    si $Ref_r \notin dom(E'_F)$ **alors**
      $E_{F'} \longleftarrow E_{F'} \cup (Ref_r, Val)$***(6)***
    sinon
      $soit(Ref_r, I') \in E'_F : C \longleftarrow C \cup (I' = Val)$***(6)***
    si $statut_F = clos$ *ou* $statut_{F'} = clos$ **alors**
      $statut_F \longleftarrow clos$;
      $statut_{F'} \longleftarrow clos$ ***(8)***
fin

5.3.2 Accès

Soit la contrainte $(Ref_r, Val) \in F$. Elle signifie que Val est la valeur associée par F à l'adresse référencée par Ref_r. Si cette contrainte figure dans le système de contraintes, il faut aussi que la variable référence Ref_r ne vaille pas *null* car, comme on modélise une sémantique sans erreurs, l'accès ne peut pas se faire via une référence nulle.

L'algorithme de filtrage de l'opérateur d'apparte-

Algorithme 2 : Algorithme de filtrage de l'opérateur d'accès

soient :
$(Ref_r, Val) \in F$ la contrainte à traiter
C le reste du système de contraintes
début
  pour chaque $(a, I) \in E_F$ **faire**
    si $dom(I) \cap dom(Val) = \emptyset$ **alors**
      $C \longleftarrow C \cup (Ref_r \neq a)$ ***(1)***
  si $statut_F = clos$ **alors**
    $dom(Ref_r) \longleftarrow dom(Ref_r) \cap dom(E_F)$ ***(2)***
  si $ground(Ref_r)$ **alors**
    si $Ref_r \notin dom(E_F)$ **alors**
      $E_F \longleftarrow E_F \cup (Ref_r, Val)$***(3)***
    sinon
      $soit(Ref_r, I) \in E_F : C \longleftarrow C \cup (I = Val)$ ***(3)***
fin

nance (algorithme 2) procède de la manière suivante.

1. Pour tout élément a du domaine de Ref_r tel qu'il existe (a, I) dans F, si la contrainte $Val = I$ n'a pas de solution a doit être retiré du domaine de Ref_r.
2. Si le statut de F est clos, le domaine de Ref_r doit être inclus dans celui de F.
3. Si Ref_r est instanciée et vaut a, a doit être ajouté au domaine de définition de F s'il n'y est pas, et Val et l'image de a par F sont égaux.

La contrainte d'accès est réveillée si l'ensemble de couples associé à F est enrichi, si le statut de F est modifié, si le domaine d'une variable du codomaine de F bouge, ou encore si le domaine de Ref_r ou celui de Val bouge.

5.3.3 Mise à jour

Soit la contrainte $F' = F\,[Ref_r \mapsto Val]$. Elle signifie que F' est la même fonction que F hormis pour la valeur associée à l'adresse référencée par Ref_r qui vaut Val. Si cette contrainte figure dans le système de contraintes, il faut aussi que la variable référence Ref_r ne vaille pas *null*. L'algorithme de filtrage de l'opérateur de mise à jour (algorithme 3) procède de la manière suivante.

1. Tous les éléments du domaine de définition de F doivent être dans le domaine de F' et inversement.
2. Tout élément du domaine de définition de F et de F' qui n'est pas dans le domaine de Ref_r a la même image par F et par F'.
3. Pour tout élément a du domaine de Ref_r tel qu'il existe (a, I') dans F', si la contrainte $Val = I'$ n'a pas de solution a doit être retiré du domaine de Ref_r.
4. Pour tout élément a du domaine de définition de F et de F' tel que son image par F et son image par F' ne peuvent pas être égales, $Ref_r = a$. En effet, la valeur associée à a est forcément mise à jour par l'opérateur donc Ref_r doit référencer a.
5. Pour tout élément a du domaine de Ref_r tel que (a, I') appartient à F' et (a, I) appartient à F, le domaine de I' est inclus dans l'union des domaines de I et de Val.
6. Si le statut de F est clos, le statut de F' est clos aussi et inversement. En effet, F et F' ont le même domaine de définition.
7. Si le statut de F est clos, le domaine de Ref_r doit être inclus dans le domaine de définition de F.
8. Si Ref_r est instancié et vaut a, a appartient aux domaines de définition de F et de F', Val et l'image de a par F' sont égaux.

La contrainte de mise à jour est réveillée si l'ensemble de couples associé à F ou à F' est enrichi, si le statut de F ou de F' est modifié, si le domaine d'une variable du codomaine de F ou de F' bouge, ou encore si le domaine de Ref_r ou celui de Val bouge.

Algorithme 3 : Algorithme de filtrage de l'opérateur de mise à jour

soient :
$F' = F\,[Ref_r \mapsto Val]$ la contrainte à traiter
C le reste du système de contraintes
début
  pour chaque $(a, I') \in E_{F'}$ **faire**
    si $a \in dom(Ref_r)$ **alors**
      si $dom(I') \cap dom(Val) = \emptyset$ **alors**
        $C \longleftarrow C \cup (Ref_r \neq a)$ ***(3)***
    si $(a, I) \in E_F$ **alors**
      si $a \in dom(Ref_r)$ **alors**
        si $dom(I) \cap dom(I') = \emptyset$ **alors**
          $C \longleftarrow C \cup (Ref_r = a)$***(4)***
        sinon
          $dom(I') \longleftarrow$
          $dom(I') \cap (dom(I) \cup dom(Val))$***(5)***
      sinon
        $C \longleftarrow C \cup (I = I')$ ***(2)***
    si $a \notin (dom(E_F)$ **alors**
      si $a \notin dom(Ref_r)$ **alors**
        $E_F \longleftarrow E_F \cup (a, I')$ ***(1,2)***
      sinon
        $E_F \longleftarrow E_F \cup (a, \_)$ ***(1)***
  pour chaque $(a, I) \in E_F$ **faire**
    si $a \notin dom(Ref_r)$ **alors**
      $E_{F'} \longleftarrow E_{F'} \cup (a, I)$ ***(1,2)***
    sinon
      $E_{F'} \longleftarrow E_{F'} \cup (a, \_)$ ***(1)***
  NB : à ce stade $dom(E_f) = dom(E_{F'})$
  si $statut_F = clos$ *ou* $statut_{F'} = clos$ **alors**
    $statut_F \longleftarrow clos$; ***(6)***
    $statut_{F'} \longleftarrow clos$; ***(6)***
    $dom(Ref_r) \longleftarrow dom(Ref_r) \cap dom(E_F)$ ***(7)***
  si $ground(Ref_r)$ **alors**
    si $Ref_r \notin dom(E_F)$ **alors**
      $E_F \longleftarrow E_F \cup (Ref_r, \_)$***(8)***
      $E_{F'} \longleftarrow E_{F'} \cup (Ref_r, Val)$***(8)***
    sinon
      $soit(Ref_r, I') \in E'_F : C \longleftarrow C \cup (I' = Val)$***(8)***
fin

Illustration *On considère un système de contraintes qui comprend les variables modélisant des fonctions F et F', la variable référence Ref_r, et la variable entière Val. Voici les domaines de ces variables à un certain stade de la propagation : concernant F, $E_F = \{(1,U),(2,V)\}$, $dom(U) = 0..10$, $dom(V) = 1..3$, le statut de F est non clos ; concernant F', $E_{F'} = \{(1,A),(2,B),(3,1)\}$, $dom(A) = inf..sup$, $dom(B) = 15..sup$, le statut de F' est clos ; $dom(Ref_r) = \{1,2\}$; $dom(Val) = 10..40$. Les déductions que l'on peut faire grâce à la contrainte $F' = F\,[Ref_r \mapsto Val]$ sont les suivantes :*

- *V et B sont associés à 2 respectivement par F et F', mais leurs domaines sont disjoints. D'après la quatrième règle, Ref_r doit référencer 2.*
- *U et A sont associés à 1 respectivement par F et F', et 1 n'est dans le domaine de Ref_r d'après la déduction précédente. D'après la deuxième règle, $U = A$ d'où $dom(U) = dom(A) = 0..10$.*
- *3 appartient au domaine de définition de F' mais pas à celui de F. D'après la première règle, il faut le rajouter dans le domaine de F ce qui est possible car F n'est pas clos donc E_F peut être enrichi. De plus, comme 3 n'est pas dans le domaine de Ref_r, 3 doit avoir la même image par F et par F' d'après la deuxième règle. On obtient $E_F = \{(1,U),(2,V),(3,1)\}$.*
- *Le statut de F' est clos donc le statut de F est clos aussi d'après la sixième règle.*
- *Comme le statut de F est clos, d'après la septième règle, le domaine de Ref_r doit être inclus dans le domaine de définition de F, ce qui est le cas car Ref_r vaut 2.*
- *D'après la huitième règle, B et Val sont égaux, d'où, en intersectant leurs domaines, $dom(B) = dom(Val) = 15..40$.*

Voici résumés les domaines des variables après avoir appliqué les règles de déductions de l'opérateur de mise à jour : concernant F, $E_F = \{(1,U),(2,V),(3,1)\}$, $dom(U) = 0..10$, $dom(V) = 1..3$, le statut de F est clos ; concernant F', $E_{F'} = \{(1,A),(2,B),(3,1)\}$, $dom(A) = 0..10$, $dom(B) = 15..40$, le statut de F' est clos ; $dom(Ref_r) = \{2\}$; $dom(Val) = 15..40$.

Si l'opérateur de mise à jour était exprimé sous la forme de deux contraintes : $F' = F'' \sqcup \{(Ref_r, V)\}$ et $F'' = F \setminus \{(Ref_r, V_{init})\}$, on ne pourrait pas utiliser la déduction de la règle 4. En effet, comparer les codomaines de F et F' pour réduire le domaine de Ref_r ne peut pas se faire avec ces contraintes. En guise d'illustration, reprenons l'exemple ci-dessus et les domaines des variables avant déductions, sachant que le domaine de V_{init} est inconnu. La première contrainte ne permet pas de déduire la valeur de Ref_r, la seule déduction possible est que $(3,1) \in E_{F''}$. La deuxième contrainte ne permet pas plus de déductions sur la valeur de Ref_r, de même on a simplement $(3,1) \in E_F$. L'opérateur de mise à jour permet donc plus de déductions que ces deux contraintes.

Concernant le codomaine, celui-ci est une union d'ensembles distincts. En fait, les images sont des variables typées. N'importe quel type peut-être utilisé sous réserve que certaines opérations, utilisées dans les algorithmes de filtrage des opérateurs, soient définies. Ces opérations sont l'intersection de domaines, l'union de domaines, et l'unification de deux variables. L'intersection des domaines de deux variables de types différents est vide. Dans notre modèle, comme les zones de la mémoire allouées ne sont jamais désallouées, à une même adresse du tas est toujours associée une variable de même type même si sa valeur peut changer.

6 Travaux connexes

Le modèle décrit ici est, à notre connaissance, le premier modèle à contraintes proposé pour représenter le bytecode Java. Ce travail est à rapprocher de celui de modélisation des instructions du C et du C++ réalisé dans le prototype InKa, également pour le test de logiciels [2]. Dans le cadre du test structurel de logiciel, la majorité des travaux ne considère la modélisation du programme que pour un chemin donné du programme à la fois (méthode de l'exécution symbolique) [13, 12]. Les valeurs des références locales à la méthode sont connues sur un chemin donné, leur modélisation ne nécessite pas des variables. Par contre le problème de la forme de la mémoire en entrée se pose. Dans les outils Pathcrawler [13] et Cute [12], en l'absence de préconditions indiquant la forme de la mémoire, les paramètres sont considérés comme distincts : les références en entrée de méthode sont considérées comme pointant vers des zones différentes de la mémoire (pas d'alias) et sont donc des constantes. Quant aux boucles comportant des allocations dynamiques, le chemin choisi donne le nombre de fois que la boucle est dépliée : la quantité de zones de la mémoire allouées par les boucles est donc connue. Dans [10], l'utilisation de CHRs permet d'automatiser la recherche de données de test pour des programmes en bytecode Java. Ce travail se distingue de celui présenté dans cet article de plusieurs manières. D'une part, par le domaine d'application : dans [10], il s'agit de test de conformité vis-à-vis d'une spécification, alors que dans le présent article, il s'agit de test structurel visant une couverture du code. D'autre part, par la technique utilisée : la méthode présentée dans [10] utilise des CHRs, alors que la méthode présentée dans cet article exploite la propagation de contraintes sur domaines finis pour définir des opérateurs modélisant l'accès et la mise à jour d'une mémoire.

Concernant les opérateurs entre fonctions, les travaux les plus proches sont, d'une part, ceux qui traitent de contraintes entre des ensembles, car les fonctions sont vues ici comme des ensembles de couples, et, d'autre part, les travaux qui traitent de contraintes entre des séquences ou des listes.
Pour les solveurs ensemblistes, citons Conjunto [6], Cardinal [3], CLPS-B [9] ou encore celui de Gecode [1]. Les opérateurs que nous avons proposés ici ne peuvent être implémentés directement à l'aide de ces solveurs. Premièrement, les éléments que comportent les ensembles sont composés, ce sont des couples *(antécédent,image)*, dont l'image peut être elle-même une variable. Deuxièmement, ce ne sont pas les couples *(antécédent,image)* qui doivent être distincts mais seulement les antécédents. Troisièmement, le cardinal des ensembles n'est pas connu initialement. Dans Conjunto, Cardinal ou dans le solveur ensembliste de Gecode, les ensembles peuvent être instanciés ou variables, mais ne peuvent pas contenir de variables. Par exemple, le solveur ensembliste de Gecode traite uniquement les ensembles d'entiers. Dans CLPS-B, les ensembles de variables sont traités mais leur cardinalité doit être connue.
La nécessité d'avoir des antécédents distincts dans les ensemble de couples *(antécédent,image)* rapproche nos opérateurs des opérateurs sur les listes ou sur les séquences. En effet, celles-ci peuvent être vues comme des fonctions qui à chaque index (position dans la liste ou dans la séquence) associent une variable. On peut ainsi citer la contrainte globale *element* qui permet de contraindre la valeur d'un élément situé à un certain index dans une liste. Elle est disponible dans beaucoup de solveurs (Sicstus Prolog, Eclipse Prolog, ...) et est décrite dans le catalogue de contraintes globales de Beldiceanu [4]. Cependant, la liste passée en paramètre de la contrainte *element* doit être de longueur fixée. Or nous avons besoin d'opérateurs sur des fonctions dont la taille du domaine de définition est elle aussi variable, cette spécificité exclut l'utilisation d'une telle contrainte. Il n'existe pas à notre connaissance d'opérateurs sur les listes ou les séquences supportant des listes ou des séquences de taille variable, et permettant d'exprimer nos opérateurs d'union disjointe, d'appartenance et de mise à jour.

7 Conclusion

Une modélisation pour une partie des instructions du bytecode Java a été présentée, elle est utile pour résoudre des problèmes de test logiciel. Afin de modéliser le tas, des variables représentant des fonctions définies sur une partie des entiers naturels ont été introduites, de même que le fonctionnement d'opérateurs à contraintes ayant comme paramètres de telles variables. Ces opérateurs et la traduction des instructions bytecode en contraintes sont en cours d'implémentation. Les travaux à venir porteront, d'une part, sur la modélisation des instructions de transfert de contrôle, et, d'autre part, sur la prise en compte dans notre modèle du système de type de Java. Les instructions de transfert de contrôle sont des instructions de saut dans le code, avec ou sans comparaisons. Les méthodes proposées dans [2] pour traiter les constructions conditionnelles et les boucles d'un langage impératif devront être adaptées au cas d'un code destructuré. D'autre part, modéliser le système de type de Java pourrait aider aux déductions, comme, par exemple, en présence d'appels de méthodes polymorphes.

Références

[1] http ://www.gecode.org/.

[2] A.Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. volume 1861 of *Lecture Notes in Computer Science*, 2000.

[3] F. Azevedo. Cardinal : A finite sets constraint solver. *Constraints*, 12(1) :93–129, 2007.

[4] N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Technical Report T2005-08, Swedish Institute of Computer Science, June 2005.

[5] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Softw. Test, Verif. Reliab*, 16(2) :97–121, 2006.

[6] C.Gervet. Interval propagation to reason about sets : Definition and implementation of a practical language. *Constraints*, 1(3) :191–244, 1997.

[7] F. Charreteur, B. Botella, and A. Gotlieb. Modelling dynamic memory management in constraint-based testing. In *TAIC-PART (Testing : Academic and Industrial Conference)*, Windsor, UK, 2007.

[8] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test-data generation. *IEEE Transactions on Software Engineering*, 17(9) :900–910, September 1991.

[9] F.Bouquet, B. Legeard, and F. Peureux. CLPS-B - A constraint solver to animate a B specification. *STTT*, 6(2) :143–157, 2004.

[10] S.-D. Gouraud and A. Gotlieb. Using CHRs to generate functional test cases for the Java card virtual machine. In *PADL '06 : Proc. 8th Intl. Symp. Practical Aspects of Declarative Languages*, pages 1–15, 2006.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, 1999.

[12] K. Sen, D. Marinov, and G. Agha. CUTE : a concolic unit testing engine for C. In *Proc. of the 10th European Software Engineering Conference, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005.

[13] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In *EDCC-5, 5th European Dependable Computing Conf., Budapest, Hungary, April 20-22, 2005, Proc.*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.